

Introduction

(PAGE 1) Hello everyone, you may call us the “JJC” group and today, we are going to talk about the design pattern called prototype.

(PAGE 2) Here you can see, this is our table of contents and this is where you can tell what to expect in this presentation.

(WHAT IS PROTOTYPE PATTERN) (PAGE 3) The Prototype Pattern is a creational design pattern that allows us to make new objects by copying existing objects, known as prototype. Instead of creating new objects, this pattern can clone existing instance to generate new instances with the same properties and behaviors. This can be particularly useful in situations where object creation is resource-intensive or complex, as it reduces the overhead of creating objects from scratch. By using Prototype Pattern, you can improve performance and reduce the usage of resource by reusing objects for creating new ones.

History

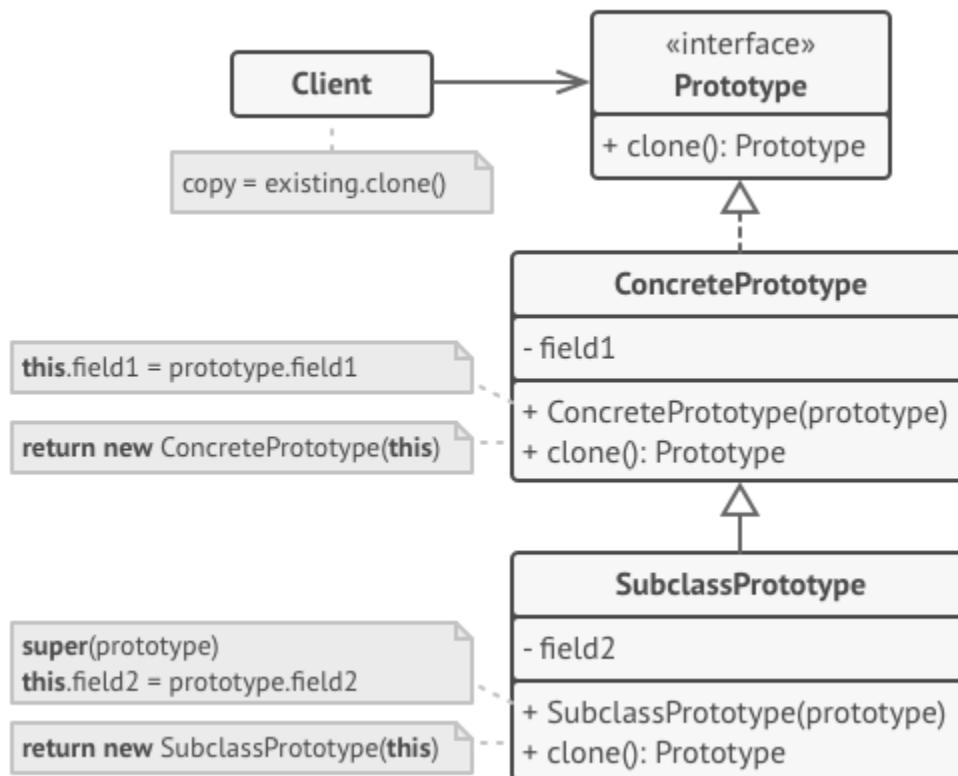
(HISTORY PAGE 1) In the late 1970s, Christopher Alexander and his colleagues came up with design patterns for building structures, emphasizing that designs should work for engineers and meet customer needs. This idea became really handy for software developers too and was embraced as a super useful tool in their community.

Back in 1964, Ivan Sutherland's Sketchpad System likely introduced the Prototype pattern. It gained traction when used in ThingLab, allowing users to create a group of objects and then turn it into a prototype by storing it in a library of reusable objects. While Goldberg and Robson acknowledged prototypes as a pattern, it was Coplien who extensively described it. In his work, he not only detailed the Prototype pattern for C++ but also provided numerous examples and different ways it could be used.

(HISTORY PAGE 2) Back in 1994, a group known as "The Gang of Four" - made up of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides - made something called the Prototype Pattern pretty famous and clear for everyone. They wrote about it in a book called "Design Patterns: Elements of Reusable Object-Oriented Software." In this book, they really explained this pattern well. They gave examples and talked a lot about how it works and how people can use it when designing object-oriented software

Definition

(DEFINITION PAGE 1) The prototype pattern is kinda like using a template to create stuff. Instead of doing things all over again, it copies something that already exists, also known as "prototype" and makes a new one just like it. This can help save more time and resource, especially when making something difficult or something that takes a lot of effort. It's handy because you can change the copied thing a bit to fit what you need without starting from scratch. So, instead of building from the ground up, you're making copies and tweaking them to get what you want.



And finally, here you can see that this is the basic structure of Prototype Pattern and here you can see the client, prototype, subclass and the concrete prototype, and so on.

Key Components

(BASIC STRUCTURE) Here are the key components and how the Prototype Design Pattern typically works:

So first, we'll start with:

- **(DEFINITION PAGE 2) Prototype:** So, this is the interface or abstract class that declares the methods for cloning itself. It includes a clone method that derived classes must implement. The clone method creates a copy of the object.
- **(DEFINITION PAGE 3) Concrete Prototype:** These are the concrete classes that implement the Prototype interface or extend the Prototype abstract class. They provide the actual implementation of the clone method, which creates a copy of the object.
- **(DEFINITION PAGE 4) Client:** The client is responsible for creating new objects by requesting the prototype to clone itself. It doesn't need to know the specific class of the object it's cloning; it just uses the clone method provided by the prototype.

Note

(DEFINITION PAGE 5) And finally, the **Subclass Prototype** which ideally, is just there to highlight that if the concrete prototype is extended it is preferable to override the clone method. Within the new implementation, we will call the super method and handle the fields introduced by the subclass. (optional->)Thank you for your attention. Now, I'd like to hand the floor over to Carl Vincent to continue our presentation.

Usages

- **Efficient Object Creation:** When object creation is more efficient by copying an existing object rather than initializing a new one from scratch.

- **Reducing Subclassing:** When a class cannot anticipate the type of objects it must create, the Prototype Pattern allows for new object creation without relying on subclasses.
- **Configuring Objects:** Allows for configuring complex objects with different properties.

Pros and Cons

Pros


- **Object Creation Efficiency:** The Prototype pattern allows you to create new objects by copying existing ones, which can be more efficient than creating objects from scratch, especially when the initialization of an object is complex or resource-intensive.
- **Flexible Object Creation:** It provides a way to create objects with different configurations or states based on existing prototypes. This flexibility is useful when you need to create variations of objects without having to subclass or modify existing code.
- **Reduced Complexity:** It simplifies the client code by abstracting the object creation process. Clients can focus on using the clone method rather than worrying about the details of object construction.
- **Maintains Object Relationships:** The Prototype pattern can help maintain relationships between objects, especially when those relationships are not trivial to establish.

Cons

- **Cloning Complexity:** Implementing the cloning logic for objects can be complex, particularly when dealing with deep copies or objects with circular references. Ensuring that all internal states are correctly copied can be error-prone.
- **Need for Proper Initialization:** Cloned objects may require additional initialization steps after cloning to be in a valid state. Failing to initialize an object properly can lead to unexpected behavior.
- **Maintaining Prototypes:** Managing a collection of prototypes can become cumbersome, especially in applications with numerous prototypes. Keeping track of and updating prototypes can add complexity to the codebase.
- **Potential for Inefficient Cloning:** In some cases, cloning objects can be less efficient than other object creation methods, especially if the cloning process is resource-intensive or if it involves deep copying of complex object graphs.

Implementation

1. Using the Cloneable Interface:




```
class ConcretePrototype implements Cloneable {  
    private String attribute;  
  
    public ConcretePrototype(String attribute) {  
        this.attribute = attribute;  
    }  
  
    @Override  
    public ConcretePrototype clone() throws CloneNotSupportedException  
    {  
        return (ConcretePrototype) super.clone();  
    }  
  
    public String getAttribute() {  
        return attribute;  
    }  
}
```

This approach closely mimics the Cloneable feature and is Java-specific.

Using the Cloneable interface in Java provides a marker to the Java Virtual Machine (JVM) that an object of the implementing class can be safely cloned using the clone() method, which is inherited from the Object class. When you implement Cloneable and override the clone() method, you're essentially telling the JVM that your object supports cloning and provides a way to create a shallow copy of it.

Without implementing the Cloneable interface and the clone() method, attempting to clone an object using the clone() method will result in a CloneNotSupportedException at runtime.

2. Custom Clone Method



```
class ConcretePrototype {  
    private String attribute;  
  
    public ConcretePrototype(String attribute) {  
        this.attribute = attribute;  
    }  
  
    public ConcretePrototype clone() {  
        return new ConcretePrototype(this.attribute);  
    }  
  
    public String getAttribute() {  
        return attribute;  
    }  
}
```

In a language-agnostic approach, you can create a custom method for copying objects without relying on language-specific features like Cloneable.

This approach is not limited to Java and can be used in other programming languages as well.

Both of these approaches allow you to implement the Prototype Pattern for object cloning, but the second approach (custom copy method) is not tied to Java.

Why use the custom clone method when you are still using the new keyword?

1. **Customized Cloning Logic:** Implementing a custom clone method allows you to define the exact cloning behavior that makes sense for your class. You have full control over how objects are copied, which can be crucial for complex or specialized classes.
2. **Flexibility:** You are not limited by the default shallow cloning behavior provided by the Cloneable interface. With a custom clone method, you can perform deep copies, copy additional properties, or implement any other specific logic required for cloning.
3. **Independence:** You are not tied to the conventions of the Cloneable interface, which can be liberating if you want to design your classes in a way that does not conform to this convention.

Cloning Strategies

For the Prototype Design Pattern, it is mandatory to have a copy feature in existing objects. If we want to create prototypes of our object, we need to implement/define a clone() method for that object/class. There are 2 ways to implement the clone method, a shallow copy or a deep copy.

1. Shallow Copy

When you use a shallow copy in the Prototype pattern, you are essentially copying the top-level attributes and references of the prototype object. This means that if the prototype object contains references to other objects (e.g., nested objects or sub-objects), those references are copied, but the objects themselves are not duplicated. In other words, the copied object and the prototype object will share references to the same nested objects.

In this case, if you create a shallow copy of the prototype, changes made to the nested object within the copy will also affect the nested object in the original prototype because they share the same reference.

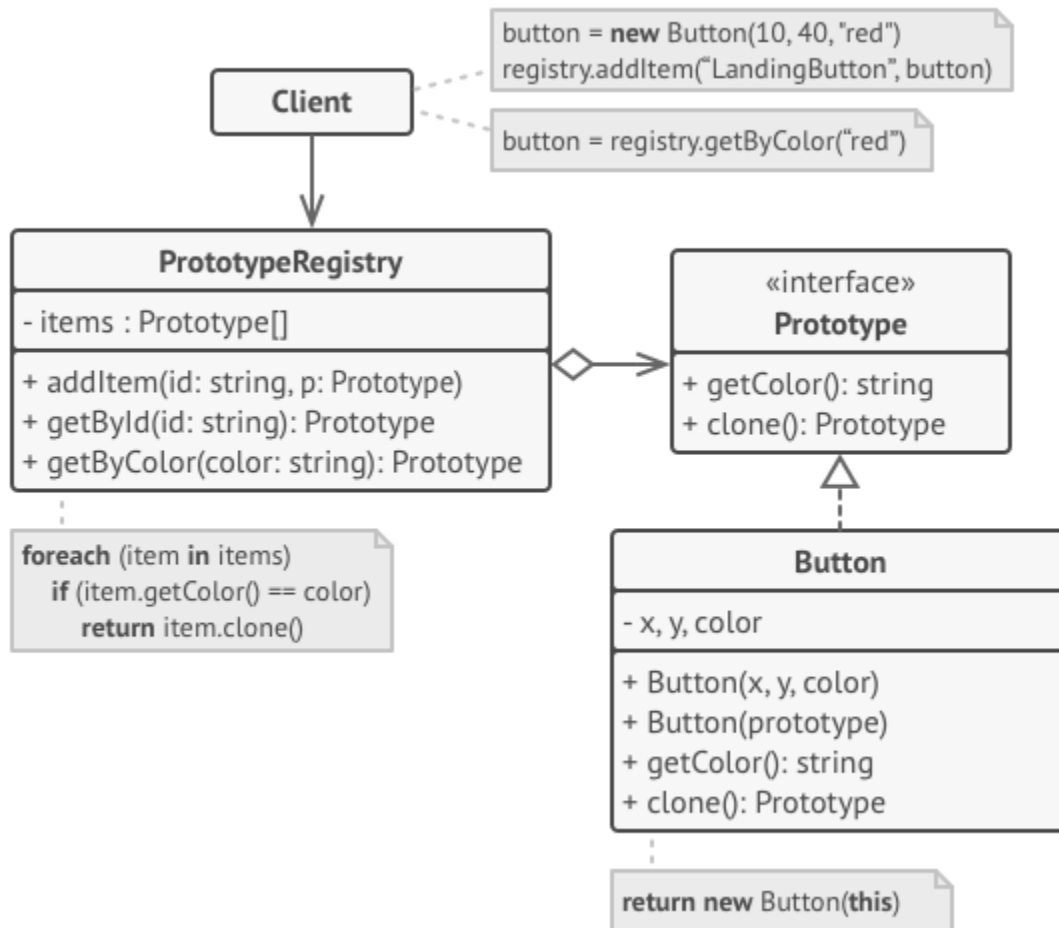
2. Deep Copy

On the other hand, if you use a deep copy in the Prototype pattern, you copy the top-level attributes and recursively copy all nested objects or sub-objects within the prototype. This ensures that the copied object is completely independent of the prototype and its nested objects.

In this case, changes made to the nested object within the copy will not affect the nested object in the original prototype because they are separate objects with their own memory.

The choice between using a shallow copy or a deep copy in the Prototype pattern depends on whether you want the cloned objects to share references with the original or be entirely independent. Shallow copies can be more efficient if shared references are acceptable, while deep copies provide complete isolation between objects but may be more resource-intensive due to the recursive copying of nested objects.

Prototype Registry



A Prototype Registry, also known as a Prototype Manager or Prototype Store, is a design pattern and architectural concept used in software engineering. It is particularly relevant in the context of the Prototype pattern.

The Prototype Registry is a central repository or store that holds a collection of pre-defined prototype objects. These prototype objects serve as templates for creating new objects with similar properties and behaviors. The primary purpose of a Prototype Registry is to provide a convenient way to access and clone frequently accessed prototype objects.

The Prototype Registry is commonly used in scenarios where object creation is resource-intensive, and having a central store of pre-configured prototypes can improve application performance and maintainability. It is often used in conjunction with the Prototype pattern to provide a more organized and centralized way of managing and accessing prototype objects.

Examples

Real World Examples

1. In game development, you can use the prototype pattern to create new game characters by copying existing character templates. This is particularly useful for creating similar characters with different attributes.
2. In graphical user interfaces, you can use the prototype pattern to create copies of GUI components (e.g., buttons, dialogs) to save time and resources when generating similar UI elements.
3. When working with databases, you can use the prototype pattern to clone database records, especially when creating new records with similar attributes

Java Code Examples

GitHub Repository: <https://github.com/johnivanpuayap/PrototypeDesignPattern>

Best Practices

1. Consider Architectural Context:
 - Evaluate whether the Prototype pattern is the right fit for your application's architectural needs. It may be more suitable for certain types of applications than others.
2. Use Cloning for Object Creation:
 - Implement cloning mechanisms to create new instances based on prototype objects.
 - Decide whether to use shallow or deep cloning based on the specific requirements and relationships between objects.
3. Prototype Registry (if needed):
 - Consider using a Prototype Registry when dealing with a large number of prototype objects or when resource efficiency is a concern.
 - Register prototype objects with the registry during application initialization for easy access.
4. Create and Document Prototype Objects:
 - Define and document prototype objects that serve as templates for creating new objects.
 - Ensure that prototype objects are clear, complete, and well-documented, describing their properties, behaviors, and intended use.

5. Clear Separation of Concerns:
 - Keep the creation and initialization logic separate from the client code that requests object clones.
 - Ensure that clients don't need to know the specifics of object creation; they should only interact with the Prototype pattern interface.
6. Encapsulation:
 - Encapsulate the details of cloning and prototype management to minimize direct client access to internal implementation details.
7. Maintain Consistency:
 - Ensure that all cloned objects derived from the same prototype exhibit consistent behavior and initial states.
8. Customization Guidelines:
 - Provide guidelines and documentation on when and how clients can customize cloned objects. Make sure customization does not violate the integrity of the design.
9. Error Handling:
 - Implement error handling for scenarios where prototype objects are unavailable or cloning fails.
10. Testing and Validation:
 - Thoroughly test the Prototype pattern implementation, including the cloning process and the behavior of cloned objects.
 - Validate that cloned objects match the expected state and behavior defined by their prototypes.
11. Documentation and Training:
 - Ensure that team members are trained in the use of the Prototype pattern and its implementation in your specific project.
 - Maintain clear and up-to-date documentation on how the pattern is applied within your codebase.

References

- [1] Refactoring Guru. "Prototype Design Pattern." [Online]. Available: <https://refactoring.guru/design-patterns/prototype>. [Accessed: September 16, 2023].
- [2] GeeksforGeeks. "Prototype Design Pattern." [Online]. Available: <https://www.geeksforgeeks.org/prototype-design-pattern/>. [Accessed: September 16, 2023].
- [3] DigitalOcean. "Prototype Design Pattern in Java." [Online]. Available: <https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java>. [Accessed: September 16, 2023].
- [4] Stack Overflow. "What's the Point of the Prototype Design Pattern?" [Online]. Available: <https://stackoverflow.com/questions/13887704/whats-the-point-of-the-prototype-design-pattern>. [Accessed: September 16, 2023].
- [5] D. H. Lorenz and J. E. Kidd. "The Concept of Patterns (in Proceedings of OOPSLA '95)." [Online]. Available: [https://cs.smu.ca/~porter/csc/465/notes/design_patterns.html#:~:text=The%20concept%20of%20patterns%20\(in,\(They%20had%20253%20patterns.\).](https://cs.smu.ca/~porter/csc/465/notes/design_patterns.html#:~:text=The%20concept%20of%20patterns%20(in,(They%20had%20253%20patterns.).) [Accessed: September 16, 2023].
- [6] University of North Carolina at Chapel Hill. "Design Patterns - Elements of Reusable Object-Oriented Software." [Online]. Available: <https://www.cs.unc.edu/~stotts/GOF/hires/pat3dfso.htm>. [Accessed: September 16, 2023].
- [7] Pentalog. "Prototype Design Pattern." [Online]. Available: <https://www.pentalog.com/blog/design-patterns/prototype-design-pattern/>. [Accessed: September 16, 2023].
- [8] Geek Culture. "Overview of Prototype Design Pattern." [Online]. Available: <https://medium.com/geekculture/overview-of-prototype-design-pattern-3eafaf006fde>. [Accessed: September 16, 2023].
- [9] "The Prototype Pattern Explained and Implemented in Java | Creational Design Patterns | Geekific."
- [10] A. Borning, "The programming language aspects of ThingLab—a constraint-oriented simulation laboratory," ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, pp. 343–387, October 1981.
- [11] I. E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," PhD thesis, MIT, 1963.

[12] J. O. Coplien, "Advanced C++ Programming Styles and Idioms," Addison-Wesley, Reading, MA, 1992.

[13] A. J. Goldberg and D. Robson, "Smalltalk-80: The Language and Its Implementation," Addison-Wesley, Reading, MA, 1983.